Rewriting Aggregate Queries using Description Logic

David DeHaan, David Toman, and Grant Weddell
School of Computer Science, University of Waterloo
{dedehaan, david, gweddell}@uwaterloo.ca

Abstract

This paper presents an application of a DL reasoner to the optimization of an object-relational query language. Queries containing aggregate functions are difficult to optimize because care must be taken to guarantee that the output value of the aggregate function is not affected. We present a mapping from an object-relational aggregate query to a DL implication problem such that satisfaction of the implication is a sufficient condition for the correctness of a particular logical rewrite rule with respect to a given schema.

1 Introduction

An aggregate function is a function that takes as input a bag of values and outputs a single value. An aggregate query is a query that contains an aggregate function. Aggregate functions may occur frequently in certain application domains; for example, query workloads in OLAP and data warehousing environments make substantial use of aggregation [3, 8]. The most commonly discussed aggregate functions are those defined by SQL—`Sum`, `Average`, `Maximum`, `Minimum`, and `Count`. However, many other aggregates are possible: `Median`, `Standard Deviation`, `Parity`, and `Product`, just to name a few with mathematical significance.

Description Logics can be used as a representation for the preconditions of the rules of a query optimizer, as well as for the constraints of an object-relational database schema. One advantage of this approach is that all of the reasoning about the applicability of optimization rules can be reduced to instances of logical implication. The rest of this paper describes a particular query rewrite rule for aggregate queries and how Description Logics with an ability to express uniqueness constraints can be used to reason about the applicability of the rule. In particular, this paper contributes three sufficient conditions—formulated as DL implication problems—for the commuting of a join operator with an aggregation operator which handle cases missed by algorithms that assume the aggregation operator occurs at the root of the query. This builds directly upon the work in [9, 10].

2 Definitions

In this section we define a Description Logic called $\mathcal{DLFDE}$ and an object-relational query language called $\mathcal{QLA}$. Both of these languages are extensions of similar languages found in [10].

$$
\begin{array}{lll}
D ::= & C & (D)^{\mathcal{I}} = \quad (C)^{\mathcal{I}} \subseteq \Delta \\
\mid & \top & \Delta \\
\mid & \bot & \emptyset \\
\mid & D_1 \sqcap D_2 & (D_1)^{\mathcal{I}} \cap (D_2)^{\mathcal{I}} \\
\mid & D_1 \sqcup D_2 & (D_1)^{\mathcal{I}} \cup (D_2)^{\mathcal{I}} \\
\mid & \forall f.D & \{x : (f)^{\mathcal{I}}(x) \in (D)^{\mathcal{I}}\} \\
\mid & \neg D & \Delta \setminus (D)^{\mathcal{I}} \\
E ::= & D & (E)^{\mathcal{I}} = \quad (D)^{\mathcal{I}} \\
\mid & E_1 \sqcap E_2 & (E_1)^{\mathcal{I}} \cap (E_2)^{\mathcal{I}} \\
\mid & (\mathrm{Pf}_1 = \mathrm{Pf}_2) & \{x : (\mathrm{Pf}_1)^{\mathcal{I}}(x) = (\mathrm{Pf}_2)^{\mathcal{I}}(x)\} \\
\mid & D : \{\mathrm{Pf}_1, \ldots, \mathrm{Pf}_k\} \to \mathrm{Pf} & \{x : \forall y \in (D)^{\mathcal{I}}. \bigwedge_{i=1}^{k} (\mathrm{Pf}_i)^{\mathcal{I}}(x) = (\mathrm{Pf}_i)^{\mathcal{I}}(y) \\
 & & \qquad \Rightarrow (\mathrm{Pf})^{\mathcal{I}}(x) = (\mathrm{Pf})^{\mathcal{I}}(y)\}
\end{array}
$$

Figure 1: Syntax and Semantics of $\mathcal{DLFDE}$

Definition 1 (Syntax and Semantics of $\mathcal{DLFDE}$)
Let F be a set of attribute names $\{f_1, f_2, \ldots\}$. A path expression *is defined by the grammar "$\mathsf{Pf} ::= f.\mathsf{Pf} \mid Id$" for $f \in F$. Let $\{C_1, C_2, \ldots\}$ be primitive concept descriptions. We define derived* concept descriptions *by the grammar in Figure 1. An* inclusion dependency *is an expression of the form $D \sqsubseteq E$.*
The semantics *of expressions is defined with respect to a structure $(\Delta, \cdot^{\mathcal{I}})$, where Δ is a domain of "objects" $\{e_1, e_2, \ldots\}$ and $(\cdot)^{\mathcal{I}}$ an interpretation function that fixes the interpretations of primitive concepts C to be subsets of Δ and primitive attributes f to be total functions*[1] *$(f)^{\mathcal{I}} : \Delta \to \Delta$. The interpretation is extended to path expressions, $(Id)^{\mathcal{I}} = \lambda x.x$, $(f.\mathsf{Pf})^{\mathcal{I}} = (\mathsf{Pf})^{\mathcal{I}} \circ (f)^{\mathcal{I}}$ and derived concept descriptions D and E as defined in Figure 1. An interpretation* satisfies an inclusion dependency $D \sqsubseteq E$ *if $(D)^{\mathcal{I}} \subseteq (E)^{\mathcal{I}}$. A* terminology $\mathcal{T}$ *consists of a finite set of inclusion dependencies. The* logical implication problem *asks if $\mathcal{T} \models D \sqsubseteq E$ holds; that is, if all interpretations that satisfy all constraints in $\mathcal{T}$ also satisfy $(D)^{\mathcal{I}} \subseteq (E)^{\mathcal{I}}$ (the posed question).*

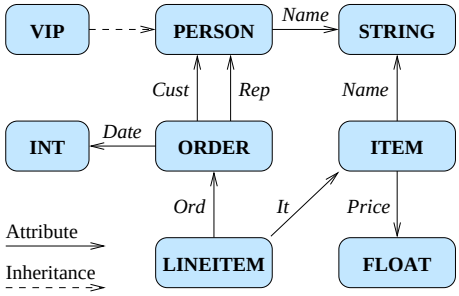
A $\mathcal{DLFDE}$ terminology can be used to represent the schema of an object-relational database; in particular, each object type corresponds to a primitive concept. In order for the implication problems to be decidable in $\mathcal{DLFDE}$, certain stratification requirements over equational constraints must be met [10]. In this case, the $\mathcal{DLFDE}$ implication problem is DEXPTIME-complete.

The following example demonstrates how a $\mathcal{DLFDE}$ terminology can be used to represent an object-relational schema. This schema will be used as a running example throughout the paper.

Example 1 Figure 2 illustrates a schema for a simple purchase order database along with the corresponding $\mathcal{DLFDE}$ terminology. The constraints in $\mathcal{T}$ induce the primitive concepts PERSON, VIP, ORDER, ITEM, LINEITEM, STRING, INT, and FLOAT. Observe that the final constraint on the ORDER concept models the fact that orders for a VIP customer are always handled by the same sales rep.

Definition 2 (Object-Relational Query Language $\mathcal{QLA}$)
Let $V \uplus B \subseteq F$ be respective sets of query variables $\{a_1, a_2, \ldots\}$ and aggregate functions

[1] DL languages are somewhat divided over the issue of whether or not attribute descriptions should denote partial or total functions. We follow [1] in opting for the latter case in order to avoid complications that would otherwise arise that relate to the semantics of some of our constructors[2].



$$
\begin{aligned}
\mathcal{T} = \{\ & \text{PERSON} \sqsubseteq (\forall \mathit{Name}.\text{STRING}),\\
& \text{VIP} \sqsubseteq \text{PERSON},\\
& \text{ITEM} \sqsubseteq (\forall \mathit{Name}.\text{STRING})\\
& \quad \sqcap (\forall \mathit{Price}.\text{FLOAT})\\
& \quad \sqcap (\text{ITEM} : \{\mathit{Name}\} \rightarrow \mathit{Id}),\\
& \text{ORDER} \sqsubseteq (\forall \mathit{Cust}.\text{PERSON})\\
& \quad \sqcap (\forall \mathit{Rep}.\text{PERSON}) \sqcap (\forall \mathit{Date}.\text{INT})\\
& \quad \sqcap (\text{ORDER} : \{\mathit{Cust}, \mathit{Date}, \mathit{Rep}\} \rightarrow \mathit{Id})\\
& \quad \sqcap (\forall \mathit{Cust}.\text{VIP} : \{\mathit{Cust}\} \rightarrow \mathit{Rep}),\\
& \text{LINEITEM} \sqsubseteq (\forall \mathit{Ord}.\text{ORDER})\\
& \quad \sqcap (\forall \mathit{It}.\text{ITEM})\ \}
\end{aligned}
$$

Figure 2: Purchase Order Database Schema

$Q ::=$	$[\![Q]\!]\mathcal{I} \equiv$
D `as` a	$\{\langle a : e\rangle : e \in (D)^{\mathcal{I}}\}$
$(a_1.\text{Pf}_1 = a_2.\text{Pf}_2)$	$\{\langle a_1 : e_1, a_2 : e_2\rangle : (\text{Pf}_1)^{\mathcal{I}}(e_1) = (\text{Pf}_2)^{\mathcal{I}}(e_2)\}$
`true`	$\{\langle\ \rangle\}$
`empty` $a_1, \ldots, a_n$	$\emptyset$
`select` $a_1, \ldots, a_n$ Q_1	$\{\!\{\langle a_1 : t@a_1, \ldots, a_n : t@a_n\rangle : t \in [\![Q_1]\!]\mathcal{I}\}\!\}$
`elim` $a_1, \ldots, a_n$ Q_1	$\{\langle a_1 : t@a_1, \ldots, a_n : t@a_n\rangle : t \in [\![Q_1]\!]\mathcal{I}\}$
`agg` $a_1, \ldots, a_n$, $\beta(a)$ `as` a_β Q_1	$\{t_1 \cup \{a_\beta : \text{AV}_{t_1}\}, \ldots, t_k \cup \{a_\beta : \text{AV}_{t_k}\} :$ $\{t_1, \ldots, t_k\} = [\![\texttt{elim}\ a_1, \ldots, a_n\ Q_1]\!]\mathcal{I}\}$ where $\text{AV}_t = \beta(\{\!\{e \in \Delta : t \cup \{a : e\} \in [\![\texttt{select}\ a_1, \ldots, a_n, a\ Q_1]\!]\mathcal{I}\}\!\})$
`from` Q_1, Q_2	$[\![Q_1]\!]\mathcal{I} \bowtie [\![Q_2]\!]\mathcal{I}$
Q_1 `union all` Q_2	$[\![Q_1]\!]\mathcal{I} \uplus [\![Q_2]\!]\mathcal{I}$

Figure 3: Syntax and Semantics of $\mathcal{QLA}$

$\{\beta_1, \beta_2, \ldots\}$.[2]

The syntax and semantics of $\mathcal{QLA}$ are shown in Figure 3. The semantics of a query Q, denoted $[\![Q]\!]$, is a function that maps interpretations to bags of tuples. Query variables occuring in a given query are assumed to satisfy standard conditions of well-formedness, and are never reused within the query. Also, for a given tuple $t = \langle a_1 : e_1, \ldots, a_n : e_n\rangle \in [\![Q]\!]\mathcal{I}$ computed by query Q over interpretation $\mathcal{I}$, we write $t@a_i$ to denote e_i.

Although the `agg` operator defined for $\mathcal{QLA}$ is restricted to taking only a single aggregate function β, this does not limit the expressive power of the language since any `agg` operator containing multiple aggregate functions can be rewritten as the natural join of multiple `agg` operators, each taking a single aggregate function. For the rest of the paper, common abbreviations to syntax will be made; e.g., `from` $Q_1, \ldots, Q_k$ stands for iterated binary joins.

3 Related Work

Testing the equivalence of arbitrary aggregate queries has been characterized for conjunctive [11], positive [6], and disjunctive queries with limited negation [5]. However,

[2] We have selected V and B as subsets of the attribute names in order to reduce the need for additional notational baggage. In particular, query variables "become" attribute names in our abstraction of queries as $\mathcal{DLFDE}$ concept descriptions.

this body of work does not consider schema information, and only considers queries where the aggregation operator is at the root of the query operator tree.

Khizder et al [9] use Description Logic to deduce the correctness (with respect to a schema) of a rewrite rule for moving subqueries out of the scope of the `elim` operator. Unfortunately, their construction requires converting the query to a normal form that is only possible because their query language is conjunctive and does not contain aggregate functions. Liu et al [10] extend this to positive queries by introducing the concept of *query context* which eliminates the reliance on a normal form.

Definition 3 (Query Context) *A* query context $Q[\,]$ *is a* $\mathcal{QLA}$ *expression containing a single instance of the special terminal symbol* $[\,]$*. For* $Q' \in \mathcal{QLA}$*, the expression* $Q[Q']$ *denotes the syntactic substitution of* Q' *for* $[\,]$*.*

Yan and Larson [12, 13] consider a rewrite that commutes a top-level aggregation operator with the join of two base relations. Let C denote the base relation pulled out of the aggregate block, V the set of attributes in the selection list of the aggregation operator, and R the attributes from C that form the join key with the rest of the aggregation block. They show that a necessary and sufficient condition for the correctness of the rewrite with respect to schema $\mathcal{T}$ (assuming an arbitrary aggregate function) is that both the functional dependencies $V \rightarrow R$ and $V \cup R \rightarrow \text{Key}(C)$ are implied by the constraints in $\mathcal{T}$ and the rest of the query. This condition has been used within the context of a system for answering queries using views [7], but it no longer remains necessary when the aggregation block is not at the root of the query operator tree. Similar rewrites have been considered in [4, 8].

4 DL Reasoning for Rewriting Aggregate Queries

In this section we show how a reasoner for $\mathcal{DLFDE}$ terminologies can be applied to rewriting aggregate queries expressed in $\mathcal{QLA}$. Specifically, we wish to generalize the rule of Yan and Larson described in the previous section to allow for the commuting of a join with an aggregation operator within general query contexts.

4.1 Generalizing Join-Aggregation Commutation

The rewrite rule of [12] can be generalized in several ways. First, the two functional dependencies in the condition can be combined into the single equivalent dependency $V \rightarrow Id_C$, where Id_C is a unique tuple identifier for tuples from relation C (follows since, by definition, $Id_C \rightarrow R$). Next, relation C can be allowed to be an arbitrary subquery. Finally, rather than considering only top-level aggregation operators, one can allow the aggregation block to occur at arbitrary positions within a larger containing query. One impact of this last generalization is that the functional dependency is no longer a necessary condition for correctness of the rewrite; it is possible for the overall queries to be equivalent without the aggregate subqueries being equivalent. This can happen when the rewritten aggregate block is nested within another aggregation operator whose aggregate function ignores or compensates for the changes introduced by the rewrite.

Example 2 The following queries are equivalent even though the dependency

$$C_Q \sqsubseteq \top : \{\mathsf{Date}\} \rightarrow o.Id$$

does not hold for all databases, where C_Q is a primitive concept corresponding to the result of the query.

$\mathcal{E}_1$: agg Sum(*ItemCnt*) as *TotalItemCnt*
[agg *Date*, Count(l) as *ItemCnt*
from ORDER as o, LINEITEM as l,
(o.*Date* = *Date*), (o = l.*Ord*)]

$\mathcal{E}_2$: agg Sum(*ItemCnt*) as *TotalItemCnt*
[select *Date*, *ItemCnt*
from ORDER as o,
(agg o, Date, Count(l) as *ItemCnt*
from LINEITEM as l,
(o.*Date* = *Date*), (o = l.*Ord*))]

The queries in Example 2 are equivalent even though the output of the [] blocks may differ. This is because the output values from the `Count` function are re-aggregated by the `Sum` function in such a way that the granularity of the inner grouping does not matter. More formally, we say that the `Sum` function is *cleanly composable* with the `Count` function.

Definition 4 (Cleanly Composable Aggregate Functions)
Aggregate functions β' and β cleanly compose *if and only if for any two bags of values M_1 and M_2,*

$$\beta'(\{\!|\beta(M_1 \uplus M_2)|\!\}) = \beta'(\{\!|\beta(M_1), \beta(M_2)|\!\}).$$

Some examples of cleanly composable pairings besides `Sum-Count` include `Sum-Sum`, `Product-Product`, `Max-Max`, and `Min-Min`.

Example 3 The following queries are minor modifications of the queries from Example 2. They are also equivalent.

$\mathcal{E}_1$: agg Max(*Date*) as *MaxDate*
[agg *Date*, Count(l) as *ItemCnt*
from ORDER as o, LINEITEM as l,
(o.*Date* = *Date*), (o = l.*Ord*)]

$\mathcal{E}_2$: agg Max(*Date*) as *MaxDate*
[select *Date*, *ItemCnt*
from ORDER as o,
(agg o, Date, Count(l) as *ItemCnt*
from LINEITEM as l,
(o.*Date* = *Date*), (o = l.*Ord*))]

The above queries are equivalent because when the join is moved out of the scope of the inner aggregation operator, any differing values for *ItemCnt* are not used and the `Max` function ignores any duplicate tuples introduced. More formally, `Max` is a *duplicate insensitive* aggregate function.

Definition 5 (Duplicate Insensitive Aggregate Function)
An aggregate function β is duplicate insensitive *if and only if for any bag of values M,*

$$\beta(M) = \beta(\{e : e \in M\}).$$

Other examples of duplicate insensitive functions include `Min` and `CountDistinct`.

We wish to design a rewrite rule that captures the scenarios illustrated by Examples 2 and 3. In order to do so, we introduce some terminology that allows us to concretely refer to an aggregation block that is "above" the current aggregation block being examined.

Definition 6 (Nearest Enclosing Aggregation Operator)
Given a query context $Q[\,]$, the nearest enclosing aggregation operator *to* $[\,]$ *(if such an operator exists) is the* `elim` *or* `agg` *operator such that* $Q[\,] = Q''[\texttt{elim}\ V'\ Q'[\,]]$ *or* $Q[\,] = Q''[\texttt{agg}\ V', \beta'(a_\beta)\ \texttt{as}\ a_{\beta'}\ Q'[\,]]$ *where* Q' *does not contain* `elim` *or* `agg`.

We are now ready to give an informal outline of our desired rewrite rule. Given queries

$\mathcal{E}_1$: $\quad Q[\texttt{agg}\ V, \beta(a)\ \texttt{as}\ a_\beta\ \texttt{from}\ Q_1, Q_2]$
$\mathcal{E}_2$: $\quad Q[\texttt{select}\ V, a_\beta\ \texttt{from}\ Q_1, (\texttt{agg}\ W, \beta(a)\ \texttt{as}\ a_\beta\ Q_2)]$

(W will be defined later), define $\mathcal{E}_1'$ and $\mathcal{E}_2'$ to be the respective contents of $[\,]$ for $\mathcal{E}_1$ and $\mathcal{E}_2$. Then, $\mathcal{E}_1 \equiv \mathcal{E}_2$ if we can determine that any one of the following conditions hold.

1. $\mathcal{E}_1' \equiv \mathcal{E}_2'$ within the context of $Q[\,]$.
2. Potentially differing values for a_β are "compensated for" by a clean composition of the aggregate function of the nearest enclosing aggregate operator with β.
3. Potentially differing values for a_β are not used, and any duplicate tuples introduced are removed by the nearest enclosing aggregate operator.

For the last two conditions, we need to ensure that the values for a_β are not used in some value-dependent manner (such as in a predicate) anywhere in the query operator tree between $[\,]$ and the nearest enclosing aggregate operator.

4.2 Expressing the Rules within $\mathcal{DLFDE}$

To utilize a DL reasoner for testing correctness of the rewrite we need to represent in $\mathcal{DLFDE}$ the constraints induced by the query structure. In preparation, it will be helpful to extend our schema information with additional knowledge about aggregate functions.

Definition 7 *We define an* aggregate schema, *written $\mathcal{T}_{agg}$, as*

$$\mathcal{T}_{agg} = \{C_\beta \sqsubseteq C_{DI} : \beta \in B \text{ and } \beta \text{ is duplicate insensitive}\}$$
$$\cup\, \{C_{\beta'} \sqsubseteq \forall\beta.C_{CC} : \beta', \beta \in B \text{ and } \beta', \beta \text{ are cleanly composable}\}$$

where C_{DI}, C_{CC}, and each C_β and $C_{\beta'}$ are distinct primitive concepts.

The aggregate schema information depends only on the aggregate functions available in the system and not on each query; therefore, it can be stored along with $\mathcal{T}$.

Now we are ready to model the query-dependent constraints for a given invocation of the DL reasoner to decide the applicability of the rewrite rule to the query. For subquery Q and query context $Q[\,]$, we introduce the inclusion dependency $C_Q \sqsubseteq$

Q	α_Q	E_Q
D as a	$\{a\}$	$(\forall a.D) \sqcap (\top : \{a\} \rightarrow Id_Q)$
$(a_1.\mathrm{Pf}_1 = a_2.\mathrm{Pf}_1)$	$\{a_1, a_2\}$	$(a_1.\mathrm{Pf}_1 = a_2.\mathrm{Pf}_1) \sqcap (\top : \{a_1, a_2\} \rightarrow Id_Q)$
true	$\{\}$	$\top$
empty V	V	$\bot$
select V Q_1	V	$E_{Q_1} \sqcap (\top : \{Id_{Q_1}\} \rightarrow Id_Q)$
elim V Q_1	V	$E_{Q_1} \sqcap (\top : \{V\} \rightarrow Id_Q)$
agg V, $\beta(a)$ as a_β Q_1	$V \cup \{a_\beta\}$	$E_{Q_1} \sqcap (\forall a_\beta.C_\beta) \sqcap (\top : \{V\} \rightarrow Id_Q)$
from Q_1, Q_2	$\alpha_{Q_1} \cup \alpha_{Q_2}$	$E_{Q_1} \sqcap E_{Q_2} \sqcap (\top : \{Id_{Q_1}, Id_{Q_2}\} \rightarrow Id_Q)$
Q_1 union all Q_2	$\alpha_{Q_1} \cup \alpha_{Q_2}$	$\mathcal{M}(E_{Q_1} \sqcup E_{Q_2})$ [3]

$Q[\,]$	$\alpha_{Q[]}$	$E_{Q[]}$	$\gamma_{Q[]}$	$A_{Q[]}$
$[\,]$	$\{\}$	$\top$	$\{\}$	$\top$
Q_1[from Q_2, $[\,]$] or Q_1[from $[\,]$, Q_2]	$\alpha_{Q_1[]} \cup \alpha_{Q_2}$	$E_{Q_1[]} \sqcap E_{Q_2}$	$\gamma_{Q_1[]}$	$A_{Q_1[]}$
Q_1[select V $[\,]$]	$\alpha_{Q_1[]} \cup V$	$E_{Q_1[]}$	$\gamma_{Q_1[]}$	$A_{Q_1[]}$
Q_1[elim V $[\,]$]	$\alpha_{Q_1[]} \cup V$	$E_{Q_1[]}$	$\{\}$	C_{elim}
Q_1[agg V, $\beta(a)$ as a_β $[\,]$]	$\alpha_{Q_1[]} \cup V$	$E_{Q_1[]}$	$\{a\}$	$C_{\mathrm{agg}} \sqcap \forall\delta.C_\beta$
Q_1[Q_2 union all $[\,]$] or Q_1[$[\,]$ union all Q_2]	$\alpha_{Q_1[]}$	$E_{Q_1[]}$	$\gamma_{Q_1[]}$	$A_{Q_1[]}$

Figure 4: Capturing Structural Constraints

$(E_Q \sqcap E_{Q[]})$, where C_Q is a primitive concept corresponding to the result of the query, E_Q is a concept inherited "up" from Q, and $E_{Q[]}$ is a concept inherited "down" from $Q[\,]$. Figure 4 defines E_Q and $E_{Q[]}$, as well as α_Q and $\alpha_{Q[]}$ which are the sets of variable names that occur in Q and $Q[\,]$, respectively. Note that each unique object identifier Id_Q is peculiar to subexpression Q and is an artifact of the translation to $\mathcal{DLFDE}$; Id_Q need not occur as an actual attribute within the object-relational system.

The second and third conditions of the outline at the end of the previous section require knowing properties of the nearest enclosing aggregate operator in $Q[\,]$. Relevant properties of the nearest aggregate operator are captured by introducing dependencies of the form $C_Q \sqsubseteq A_{Q[]}$, where $A_{Q[]}$ is defined as in Figure 4. Note that this construction requires two distinct primitive concept names C_{agg} and C_{elim} and a distinct attribute name δ. Figure 4 also defines $\gamma_{Q[]}$ which is a set containing the variable name of the input argument to the aggregate function of the nearest enclosing aggregation operator in $Q[\,]$.

Theorem 1 *The queries*

$$\begin{aligned} \mathcal{E}_1\text{:}&\quad Q[\texttt{agg } V,\ \beta(a) \texttt{ as } a_\beta \texttt{ from } Q_1,\ Q_2] \\ \mathcal{E}_2\text{:}&\quad Q[\texttt{select } V,\ a_\beta \texttt{ from } Q_1,\ (\texttt{agg } W,\ \beta(a) \texttt{ as } a_\beta\ Q_2)]\ , \end{aligned}$$

where $W = (V \cup \alpha_{Q_1}) \cap \alpha_{Q_2}$*, are equivalent with respect to* $\mathcal{T} \cup \mathcal{T}_{agg}$ *if any of the following conditions are satisfied.*

1. $\mathcal{T} \cup \{C_Q \sqsubseteq E_{Q[]} \sqcap E_{Q_1} \sqcap E_{Q_2}\} \models C_Q \sqsubseteq (C_Q : \{V \cup \alpha_{Q[]}\} \rightarrow Id_{Q_1})$
2. $a_\beta \notin \alpha_{Q[]}$, $a_\beta \in \gamma_{Q[]}$, *and*

$$\mathcal{T} \cup \mathcal{T}_{agg} \cup \{C_Q \sqsubseteq A_{Q[]}\} \models C_Q \sqsubseteq (C_{agg} \sqcap \forall\delta.\forall\beta.C_{CC})$$

[3] $\mathcal{M}$ is a mapping that removes functional dependencies and equational constraints from its arguments. This is necessary to keep E_Q a valid $\mathcal{DLFDE}$ concept.

3. $a_\beta \notin \alpha_{Q[]}$, *and*

$$\mathcal{T} \cup \mathcal{T}_{agg} \cup \{C_Q \sqsubseteq A_{Q[]}\} \models C_Q \sqsubseteq (C_{elim} \sqcup (C_{agg} \sqcap \forall \delta . C_{DI}))$$

Proof (sketch): Let $\mathcal{E}_1'$ and $\mathcal{E}_2'$ denote the contents of [ ] for $\mathcal{E}_1$ and $\mathcal{E}_2$.

Condition 1: Suppose that $\mathcal{E}_1 \not\equiv \mathcal{E}_2$. Then there exists some interpretation $\mathcal{I}$ with a fixed valuation of $\alpha_{Q[]}$ such that $[\![\mathcal{E}_1']\!]\mathcal{I} \not\equiv [\![\mathcal{E}_2']\!]\mathcal{I}$. This is only possible if the interpretation $[\![\mathcal{E}_2']\!]\mathcal{I}$ includes two distinct tuples which agree on attributes $V \cup \alpha_{Q[]}$. However, these tuples can be used to create an interpretation $\mathcal{I}$ containing only two objects $s, t \in (C_{\mathcal{E}_2'})^{\mathcal{I}}$ that agree on attributes $V \cup \alpha_{Q[]}$ but disagree on $Id_{\mathcal{E}_2'}$. Define $Q_2' = (\text{agg } W,\ \beta(a) \text{ as } a_\beta\ Q_2)$. As these objects satisfy the left-hand side of the constructed implication, the E_Q construction for the `select` and `from` operators dictates that s and t must differ on either Id_{Q_1} or $Id_{Q_2'}$. However, by the E_Q rule for the `agg` operator, disagreement on $Id_{Q_2'}$ implies disagreement on $W \subseteq (V \cup \alpha_{Q_1})$ which implies disagreement on Id_{Q_1} (because we already stated that s, t agree on $V \cup \alpha_{Q[]}$). Therefore, we have

$$\mathcal{T} \cup \{C_Q \sqsubseteq E_{Q[]} \sqcap E_{Q_1} \sqcap E_{Q_2}\} \not\models C_Q \sqsubseteq C_Q : \{V \cup \alpha_{Q[]}\} \rightarrow Id_{Q_1}.$$

Condition 2: By construction, this condition is true only when [ ] falls into the scope of an aggregation operator and the nearest such operator to [ ] is

$$Q''[\text{agg } V',\ \beta'(a_\beta) \text{ as } a_{\beta'}\ Q'[\,]]$$

such that β' and β are cleanly composable and Q' does not contain `elim` or `agg`. Consider a single tuple $t \in [\![\text{from } Q_1,\ Q_2]\!]\mathcal{I}$ corresponding to a single pair of values (Id_{Q_1}, Id_{Q_2}). Because $a_\beta \notin \alpha_{Q[]}$ (i.e. intermediate tuples are not filtered out based upon value of α_β), for every grouping of V' in

$$[\![\text{agg } V',\ \beta'(a_\beta) \text{ as } a_{\beta'}\ Q'[\mathcal{E}_1']]\!]\mathcal{I}$$

to which t contributes f_t copies of a to the aggregate operator β' (summed across all groupings of V), an identical grouping exists in

$$[\![\text{agg } V',\ \beta'(a_\beta) \text{ as } a_{\beta'}\ Q'[\mathcal{E}_2']]\!]\mathcal{I}$$

to which t also contributes f_t multiples of a (summed across all groupings of V). It follows from Definition 4 that the value of $a_{\beta'}$ for each grouping of V' is unchanged, so $\mathcal{E}_1 \equiv \mathcal{E}_2$.

Condition 3: By construction, this condition is only true when the nearest enclosing aggregation operator to [ ] is either `elim` or `agg` with a duplicate insensitive aggregate function. However, the output of this aggregation operator is not affected by any duplicate tuples introduced by the rewrite, and since $a_\beta \notin \alpha_{Q[]}$, the value of a_β is never used (including in the selection list of the aggregation operator) so a_β does not affect the output of the query. Thus, $\mathcal{E}_1 \equiv \mathcal{E}_2$.

The rewrite rule from Theorem 1 is useful for pushing aggregation operators below joins, which reduces the cost of calculating the aggregate function. It can also be used to pull aggregation operators above a join, which could be useful for making the query match the definition of a materialized aggregate view, as in [7].

Example 4 The following queries—which both report the number of orders matching each (Cust, Rep.Name) pair for VIP customers—can be reasoned equivalent by Condition 1 of Theorem 1.

$\mathcal{E}_1$: select *Customer*, *RepName*, *OCnt*
from VIP as v, (v = *Customer*),
[agg *Customer*, *RepName*, Count(o) as *OCnt*
from PERSON as p, (p.*Name* = *RepName*),
ORDER as o, (o.*Rep* = p),
(o.*Cust* = *Customer*)]

$\mathcal{E}_2$: select *Customer*, *RepName*, *OCnt*
from VIP as v, (v = *Customer*),
[select *Customer*, *RepName*, *OCnt*
from (from PERSON as p,
(p.*Name* = *RepName*)),
(agg p, *Customer*, Count(o) as *OCnt*
from ORDER as o, (o.*Rep* = p),
(o.*Cust* = *Customer*))]

The queries inside the [] are broken down into the following subqueries.

$$Q_1 = \text{from PERSON as } p, (p.\mathit{Name} = \mathit{RepName})$$
$$Q_2 = \text{from ORDER as } o, (o.\mathit{Rep} = p), (o.\mathit{Cust} = \mathit{Customer})$$

The rewrite is correct because

$$\mathcal{T} \cup \{C_Q \sqsubseteq E_{Q[]} \sqcap E_{Q_1} \sqcap E_{Q_2}\} \models C_Q \sqsubseteq (C_Q : \{\mathit{Customer}, \mathit{RepName}\} \rightarrow Id_{Q_1})$$

holds. By examining the constraints constructed for E_{Q_1}, it is fairly obvious that p functionally determines Id_{Q_1} in the result. Therefore, the crucial step in the reasoning process is to combine the inherited constraints from $E_{Q[]}$

$$C_Q \sqsubseteq \forall v.\mathsf{VIP} \sqcap (v = Customer)$$

with various equality constraints from E_{Q_2}, plus the schema constraint

$$ORDER \sqsubseteq (\forall Cust.\mathsf{VIP} : \{Cust\} \rightarrow Rep)$$

in order to conclude that

$$C_Q \sqsubseteq (C_Q : \{\mathit{Customer}\} \rightarrow p.\, Id)$$

holds. A reasoning algorithm such as the one in [12] that reasons about aggregation blocks independent of their context in the query tree would miss the constraints from $E_{Q[]}$, and so fail to find the equivalence.

The queries given in Examples 2 and 3 can be reasoned equivalent using Conditions 2 and 3, respectively.

5 Summary

We have presented an application of a DL reasoner in testing sufficient conditions for the correctness of a query optimization rule for the object-relational query language $\mathcal{QLA}$ which includes aggregation and union operators. One novel feature of the constructed implication problems is the capturing of functional dependencies over virtual object identifiers induced by the structure of the query.

Future work includes establishing completeness results for logical rewrites involving aggregation operators. Completeness with respect to the interpretation of arbitrary aggregate functions is not realistic, as one can create sets of pathological functions whose interactions adhere to patterns that would need to be explicitly represented in

the aggregate schema. We have proposed the *duplicate insensitive* and *cleanly composable* classes of aggregate functions in an attempt to generalize two common patterns. Unfortunately, our current reasoning mechanism is not complete even with respect to un-interpreted aggregate functions. One problem is that the notion of *nearest enclosing aggregation operator* and our associated construct $A_{Q[]}$ is not powerful enough to capture transitive cases of compensation between nested aggregation blocks. To do so, we would need to extend our construction to capture the entire hierarchy of aggregation operators that occur as ancestors of [] in the query tree; we would also need to stratify $\alpha_{Q[]}$ and $\gamma_{Q[]}$ so that we could identify value-dependent usage of aggregation variables at different depths in the query tree.

Acknowledgement

The authors gratefully acknowledge the National Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, and Nortel Networks for support of this research.

References

[1] A. Borgida and P. F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, pages 277–308, 1994.

[2] M. Buchheit. Refining the structure of terminological systems: Terminology = schema + views. In *Proc. 12th National Conference on Artificial Intelligence*, pages 199–204, 1994.

[3] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[4] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. 20th Int'l Conference on Very Large Data Bases (VLDB'94)*, pages 354–366, 1994.

[5] S. Cohen, W. Nutt, and Y. Sagiv. Equivalences among aggregate queries with negation. In *Proc. 20th Symposium on Principles of Database Systems (PODS 2001)*, pages 215–226. ACM Press, 2001.

[6] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. 18th Symposium on Principles of Database Systems (PODS 1999)*, pages 155–166. ACM Press, 1999.

[7] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. 2001 ACM SIGMOD Int'l Conference on Management of Data*, pages 331–342. ACM Press, 2001.

[8] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21th Int'l Conference on Very Large Data Bases (VLDB'95)*, pages 358–369, 1995.

[9] V. L. Khizder, D. Toman, and G. Weddell. Reasoning about duplicate elimination with description logic (preliminary report). In *Computational Logic (CL/DOOD 2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 1017–1032. Springer, 2000.

[10] H. Liu, D. Toman, and G. Weddell. Fine grained information integration with description logics. In *Proc. 2002 Int'l Workshop on Description Logics (DL2002)*, volume 53 of *CEUR Workshop Proceedings*, 2002.

[11] W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In *Proc. 17th Symposium on Principles of Database Systems (PODS 1998)*, pages 214–223. ACM Press, 1998.

[12] W. P. Yan and P.-Å. Larson. Performing group-by before join. In *Proc. 10th Int'l Conference on Data Engineering (ICDE'94)*, pages 89–100. IEEE Computer Society, 1994.

[13] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Int'l Conference on Very Large Data Bases (VLDB'95)*, pages 345–357, 1995.